MAlice Milestone III - Report

James Simpson Alina Boghiu

December 15, 2011

1 The Product

Our compiler takes a MAlice program as input, tokenises it, converts it into an intermediate form and produces Intel IA-32 assembly code. It then assembles this to form an executable for the Intel platform.

We deduced the language specifications by using the examples provided, by testing our own code and of course by making use of our programming skills and intuition.

2 The Design Choices

2.1 Language

Our initial choice for writing the MAlice compiler was C++ and the *yacc* and *lex* tools. Although we succeeded in building a fully functional parser for the language, it then proved to be a bad decision in terms of code complexity. Therefore, after Milestone II we decided to radically change our approach and use Haskell. Its pattern matching features along with Happy and Alex, helped us successfully generate assembly code for MAlice.

2.2 The Lexer & Parser

To generate the Lexer and Parser, we used *Happy* and *Alex* for Haskell. Alex generates a stream of tokens which represent the text of the program and Happy translates these tokens into a list of assembly-like operations by means of the BNF-like specification we provided it. This representation is then passed to the type checker.

2.3 Type Checking

Our compiler checks for undefined variables, multiple declarations within the same scope of a variable, as well as assignment of an incorrect type to a declared variable. For this it makes use of a symbol table and assumes that a variable, once declared, is in scope for the rest of the program, but not inside a function declaration. These are of course considered separately and any variable declared here is only in scope within the body of the function.

Inside the main file, the buildSymbolTable creates a symbol table and does all the type checking at compile time. However only one action can be performed inside the "do" statement and this line has been commented out.

2.4 Code Generation

The compiler uses the intermediate representation which is output by the parser to generate assembly code. We originally planned to use the LLVM bindings for Haskell to generate LLVM IR code which could then be compiled by the LLVM compiler but xxxxxxxxxxxxxxxxxx

Functions such as reading data from the console are accomplished by using the `stdio.h` C library of assembly functions which is linked with the MAlice assembly by the `compile` command.

2.5 Operation

The compiler is operated by a bash script which takes the input alice file as its only argument. This script is run by the command `./compile`.

3 Further Development

In order to meet the deadline we had to focus on main features, but there are a few aspects which we are planning to improve on:

- Optimisation must be implemented for register allocation.
- Improve, compile and run the generated Assembly code
- Increase the number of errors caught at compile time.
- Add restrictions to the BNF